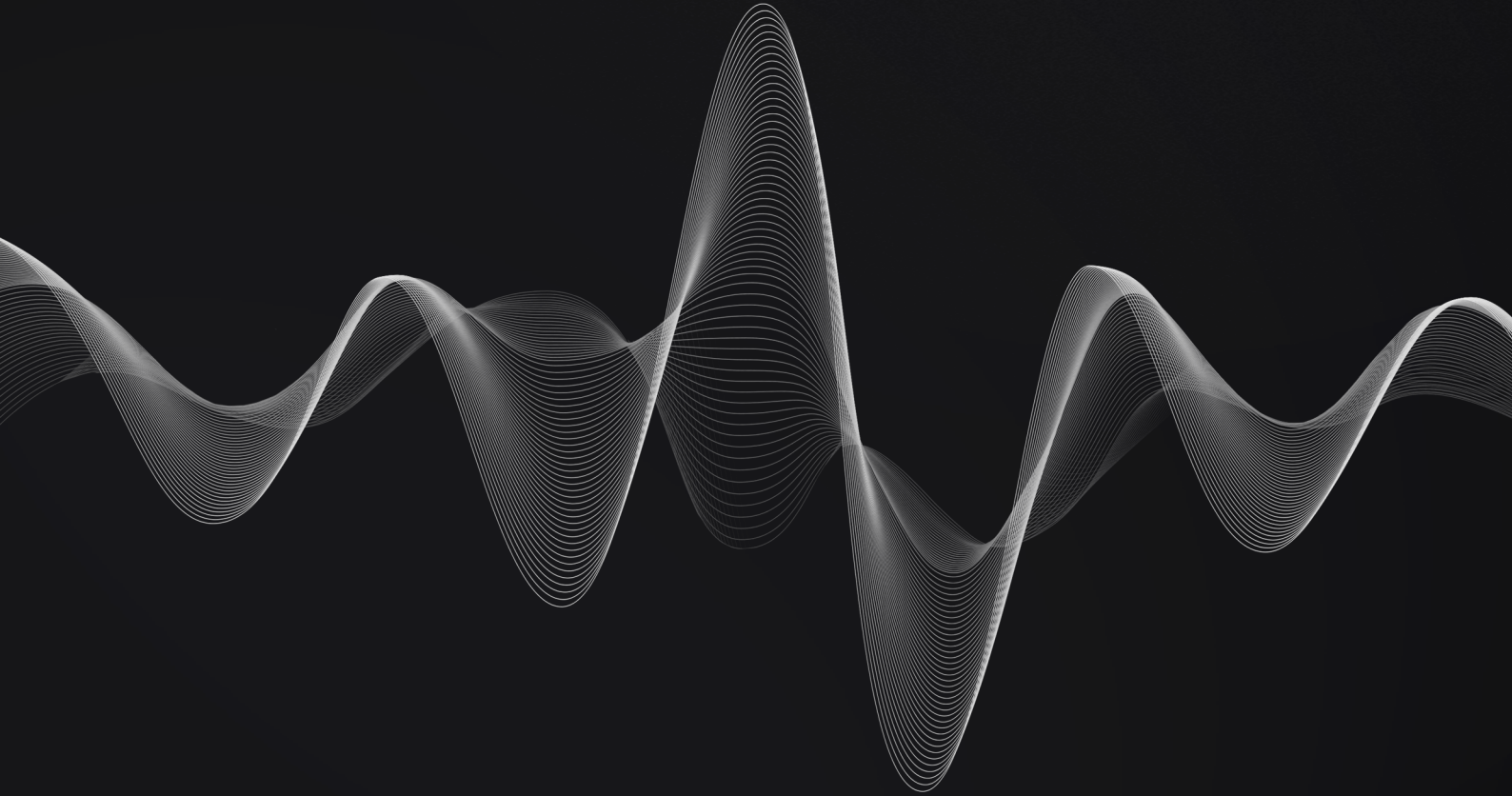




Hemi

VeHemi Smart Contract Audit Report









Document Control

PUBLIC

FINAL(v2.1)

Audit_Report_HEMI-VOT_FINAL_21

Jul 28, 2025		v0.1	Luis Arroyo: Initial draft
Jul 28, 2025		v0.2	Luis Arroyo: Added findings
Jul 28, 2025		v1.0	Charles Dray: Approved
Jul 31, 2025		v1.1	Luis Arroyo: Reviewed findings
Aug 8, 2025		v2.0	Charles Dray: Finalized
Oct 10, 2025		v2.1	Charles Dray: Published

Points of Contact	Max Sanchez	Hemi	max@hemi.xyz
	Charles Dray	Resonance	charles@resonance.security
Testing Team	Luis Arroyo	Resonance	luis.arroyo@resonance.security
	João Simões	Resonance	joao@resonance.security
	Ilan Abitbol	Resonance	ilan@resonance.security

Copyright and Disclaimer

© 2025 Resonance Security, Inc. All rights reserved.

The information in this report is considered confidential and proprietary by Resonance and is licensed to the recipient solely under the terms of the project statement of work. Reproduction or distribution, in whole or in part, is strictly prohibited without the express written permission of Resonance.

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

Contents

1 Document Control	2
Copyright and Disclaimer	2
2 Executive Summary	4
System Overview	4
Repository Coverage and Quality.....	4
3 Target	6
4 Methodology	7
Severity Rating.....	8
Repository Coverage and Quality Rating.....	9
5 Findings	10
Wrong voting Power Calculation For Expired Delegations.....	11
Usage Of ecrecover Instead Of ECDSA.recover.....	12
Initializers Could Be Front-Run	13
_delegate Function Does Not Emit Information	14
Unnecessary Initialization Of Variables With Default Values	15
A Proof of Concepts	16

Executive Summary

Hemi contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between July 21, 2025 and July 28, 2028. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 6 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Hemi with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



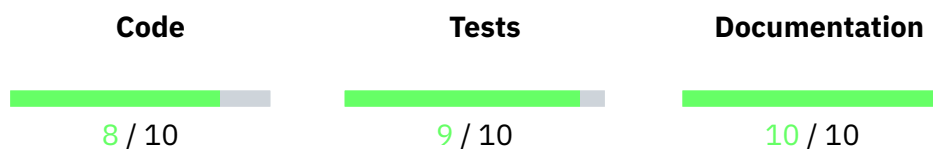
System Overview

The `VeHemi.sol` contract is a vesting and yield system based on Curve's `veCRV` and AERO voting escrow mechanism. It allows users to lock their `HEMI` tokens for up to 4 years in exchange for a NFT representing their locked position. The amount of `veHEMI` a user has determines their voting power and yield in the system.

The `VeHemiVoteDelegation` contract enables `veHEMI` token holders to delegate their voting power to other token holders without transferring ownership of the underlying tokens. Delegations take effect at the next epoch (next day boundary) and expire when the delegator's lock expires.



Repository Coverage and Quality



Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is, good**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 90%. Overall, **tests coverage and quality is excellent**.

- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent.**

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: [hemilabs/veHEMI](#)
- Hash: be7a578a80ee9416ee97d064357543a379f3dca9

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info



Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance’s Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
-|.. **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ...||| **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01		Wrong voting Power Calculation For Expired Delegations	...	Resolved
RES-02		Usage Of ecrecover Instead Of ECDSA.recover	Resolved
RES-03		Initializers Could Be Front-Run	Resolved
RES-04		_delegate Function Does Not Emit Information	Resolved
RES-05		Unnecessary Initialization Of Variables With Default Values	Resolved



Wrong voting Power Calculation For Expired Delegations

Critical

RES-HEMI-VOT01

Data Validation

Resolved

Code Section

- [src/VeHemiVoteDelegation.sol#L456](#)

Description

The vote delegation contract does not properly account for the fact that a delegator's voting power may have already been fully delegated and decayed, so when re-delegating, it can be "double-counted" or "phantom-counted" when assigned again. The subtraction from the previous delegate and addition to the new delegate is not always symmetric or accurate after time has passed and the lock has decayed.

This can be replicated in this scenario:

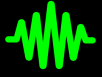
- user1, user2, and alice each create a lock (get tokenId1, tokenId2, aliceTokenId).
- user1 delegates tokenId1 to aliceTokenId. Alice now has her own votes + user1's votes.
- user1 increases unlock time for tokenId1 (from 1 year to 3 years).
- After 2 years, user1 delegates tokenId1 to user2's tokenId2.
- At this point, user1's voting power should be zero (since it was already delegated to alice and not reclaimed), but after the new delegation, both user2 and alice have more voting power than they should - sometimes more than 100% of the total.

Recommendation

When re-delegating, ensure that only the current, unclaimed voting power is moved, not the original or "normalized" values. It is also recommended to calculate the actual available voting power at the current timestamp, and only move that amount and implement checks to prevent moving zero or negative voting power.

Status

The issue has been fixed in f846e6e571467c7213fe0400cb7b3cf6baae90e2.



Usage Of ecrecover Instead Of ECDSA.recover

Low

RES-HEMI-VOT02

Data Validation

Resolved

Code Section

- [src/VeHemiVoteDelegation.sol#L120](#)

Description

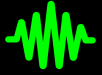
Signature Malleability is a vulnerability that can occur when improperly utilizing ecrecover function. The vulnerability allows an attacker to change the signature slightly without invalidating the signature itself.

Recommendation

It is recommended to use a well-tested library like OpenZeppelin's `ECDSA.sol` that already implements a secure signature validation process.

Status

The issue has been fixed in [f846e6e571467c7213fe0400cb7b3cf6baae90e2](#).



Initializers Could Be Front-Run

Low

RES-HEMI-VOT03

Transaction Ordering

Resolved

Code Section

- `src/VeHemi.sol#L63`

Description

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment.

Recommendation

Since the initialization function is a regular function there is a risk of it being frontrun by another transaction. If this happens, the proxy contract must be redeployed. To prevent this, the `Proxy` contract constructor makes a call to the implementation at deploy time. The initialization call must be made at this moment, encoded in the `_data` variable.

Status

The issue has been fixed in `f846e6e571467c7213fe0400cb7b3cf6baae90e2`.



_delegate Function Does Not Emit Information

Info

RES-HEMI-VOT04

Code Quality

Resolved

Code Section

- [src/VeHemiVoteDelegation.sol#L375](#)

Description

The function does not emit an event for delegation changes. Emitting an event would improve transparency and off-chain tracking.

Recommendation

It is recommended to add an event for delegation changes for better off-chain tracking.

Status

The issue has been fixed in [f846e6e571467c7213fe0400cb7b3cf6baae90e2](#).



Unnecessary Initialization Of Variables With Default Values

Info

RES-HEMI-VOT05

Gas Optimization

Resolved

Code Section

- [src/VeHemi.sol#L69](#)
- [src/VeHemi.sol#L346](#)
- [src/VeHemi.sol#L349](#)
- [src/VeHemi.sol#L374](#)
- [src/VeHemi.sol#L402](#)
- [src/VeHemi.sol#L432](#)
- [src/VeHemi.sol#L433](#)
- [src/VeHemi.sol#L720](#)

Description

In the Solidity programming language, all variables are automatically initialized to a default value corresponding to their type when they are declared. For example, integer types are initialized to 0, boolean types to `false`, and address types to `0x00`. Explicitly initializing variables to these default values when they are declared is therefore redundant, and since each operation in a contract costs gas, it results in unnecessary gas costs. This could potentially impact the contract's efficiency and the cost of executing its functions.

Several instances of this issue are found across the code base.

Recommendation

It is recommended to review the smart contract's code for variable declarations where variable are being explicitly initialized to the type's default value.

Status

The issue has been fixed in [f846e6e571467c7213fe0400cb7b3cf6baae90e2](#).

Proof of Concepts

RES-01 Wrong voting Power Calculation For Expired Delegations

Added lines:

```
function testdelegateTransferDelegate() public {
    address user1 = address(0x111111);
    address user2 = address(0x222222);

    hemiToken.mint(user1, 1_000 ether);
    hemiToken.mint(user2, 1_000 ether);

    vm.prank(user1);
    hemiToken.approve(address(veHemi), type(uint256).max);
    vm.prank(user2);
    hemiToken.approve(address(veHemi), type(uint256).max);

    uint256 amount = 100 ether;
    uint256 firstLockDuration = 2 * 365 days;
    uint256 newLockDuration = 3 * 365 days;

    (uint256 tokenId1, ) = _createLock(user1, amount, firstLockDuration);
    (uint256 tokenId2, ) = _createLock(user2, amount, newLockDuration);
    (uint256 aliceTokenId, ) = _createLock(ALICE, LOCK_AMOUNT, MAX_TIME);

    uint256 votes1 = hemiVoteDelegation.getVotes(tokenId1, user1);
    uint256 votes2 = hemiVoteDelegation.getVotes(tokenId2, user2);
    uint256 votesA = hemiVoteDelegation.getVotes(aliceTokenId, ALICE);

    _delegateAndWarp(user1, tokenId1, aliceTokenId);

    assertEq(hemiVoteDelegation.getVotes(tokenId1, user1), 0, "User1 should have no
↪ votes after delegation");
    assertLt(hemiVoteDelegation.getVotes(tokenId2, user2), votes2, "User2 should have
↪ less votes after some time");
    assertGt(hemiVoteDelegation.getVotes(aliceTokenId, ALICE), votesA, "Alice should
↪ have more votes after delegation");

    vm.prank(user1);
    veHemi.increaseUnlockTime(tokenId1, newLockDuration);

    vm.warp(block.timestamp + firstLockDuration + 1);

    uint256 votes2before = hemiVoteDelegation.getVotes(tokenId2, user2);

    _delegateAndWarp(user1, tokenId1, tokenId2);

    assertEq(hemiVoteDelegation.getVotes(tokenId1, user1), 0, "User1 should have no
↪ votes for delegation");
    assertGt(hemiVoteDelegation.getVotes(tokenId2, user2), votes2before, "votes from
↪ tokenId2 has increased after delegation of 0 vote - ERROR");
```



```
    assertGt(hemiVoteDelegation.getVotes(aliceTokenId, ALICE), votes1 + votes2 +  
↪ votesA, "Alice has more power than total - ERROR");  
}
```